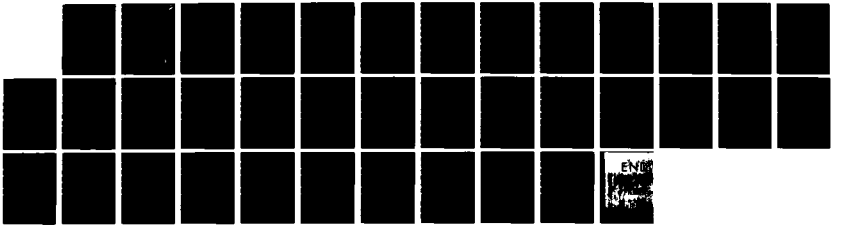
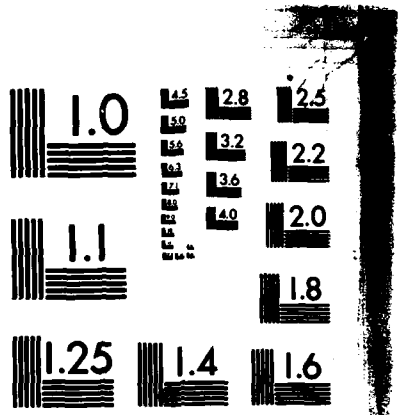
AD-A133 447 PROUST: KNOWLEDGE-BASED PROGRAM UNDERSTANDING(U) YALE 1/1
UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE
W L JOHNSON ET AL. AUG 83 YALEU/DCS/RR-285
UNCLASSIFIED N00014-82-K-0714 F/G 9/2 NL





PROUST: Knowledge-Based Program Understanding

W. Lewis Johnson and Elliot Soloway

YaleU/CSD/RR #285

August 1983

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

PROUST: Knowledge-Based Program Understanding

W. Lewis Johnson and Elliot Soloway

YaleU/CSD/RR #285

August 1983

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER #285	2. GOVT ACCESSION NO. AD A133447	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PROUST: Knowledge-Based Program Understanding		5. TYPE OF REPORT & PERIOD COVERED Technical 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) W. Lewis Johnson and Elliot Soloway		8. CONTRACT OR GRANT NUMBER(s) N00014-82-K-0714
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science Yale University New Haven, CT 06520		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 154-492
11. CONTROLLING OFFICE NAME AND ADDRESS Personnel and Training Research Programs Office of Naval Research (Code 458) Arlington, VA 22217		12. REPORT DATE August 1983 13. NUMBER OF PAGES 23
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Artificial Intelligence	Debugging Aids
Expert Systems	Programming Plans
Automatic Program Understanding	Tutoring Systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper describes a program called PROUST which does on-line analysis and understanding of Pascal programs written by novice programmers. PROUST takes as input a program and a non-algorithmic description of the program requirements, and finds the most likely mapping between the requirements and the code. This mapping is in essence a reconstruction of the design and implementation steps that the programmer went through in writing the program. A knowledge base of programming plans and strategies, together with common bugs associated with them, is used in constructing this mapping. Bugs are discovered in the process of

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

relating plans to the code; PROUST can therefore give deep explanations of program bugs by relating the buggy code to its underlying intentions.

PROUST: Knowledge-Based Program Understanding

W. Lewis Johnson
Elliot Soloway

July 1983

Yale University
Computer Science Department
New Haven, Ct. 06520
203-436-0606

This work was co-sponsored by the Personnel and Training Research Groups, Psychological Sciences Division, Office of Naval Research and the Army Research Institute for the Behavioral and Social Sciences, under Contract No. N00014-82-K-0714, Contract Authority Identification Number, Nr 154-492. Approved for public release; distribution unlimited. Reproduction in whole or part is permitted for any purpose of the United States Government.

Abstract

This paper describes a program called PROUST which does on-line analysis and understanding of Pascal programs written by novice programmers. PROUST takes as input a program and a non-algorithmic description of the program requirements, and finds the the most likely mapping between the requirements and the code. This mapping is in essence a reconstruction of the design and implementation steps that the programmer went through in writing the program. A knowledge base of programming plans and strategies, together with common bugs associated with them, is used in constructing this mapping. Bugs are discovered in the process of relating plans to the code; PROUST can therefore give deep explanations of program bugs by relating the buggy code to its underlying intentions.

1. Introduction: Motivation and Goals

Our goal is to build a tutoring system which helps novice programmers to learn how to program. This system will have two components: a *programming expert* which can analyze and understand buggy programs, and a *pedagogical expert* that knows how to effectively interact with and instruct students. We have focused our attention on the first component, with the objective of building a system that can be said to truly understand (buggy) novice programs.[1] In this paper, we will describe the theory and processing techniques by which our analysis system, PROUST, understands buggy and correct programs.

Bugs in programs are sections of code whose behavior fails to agree with the program specification. Although the presence of bugs may be indicated by various kinds of anomalous program behavior, in general bugs are not properties of programs, but rather *are properties of the relationship between programs and intentions.* [9, 10] For example, consider the program in Figure 1-1. The programmer has written a program that reads in a number and then computes the average of all the numbers between it and 99999, in integer increments. This is not what the stated problem requires; presumably the programmer was trying to solve the problem, but a bug has altered the program's behavior. How do we determine what this bug is? Note that the programmer first does a Read into the variable New, and then increments it by 1. Based on our theory of programming knowledge, [17, 12, 18, 1] we would hypothesize that the student thought that incrementing the variable New would return the next value of New; if incrementing Count gets the next INTEGER value, then incrementing New should get the next input value! The student has thus made an overgeneralization: adding one to a variable returns the next value of that variable. The key element of the above analysis is the construction of a relationship from a piece of code to a problem goal; the mechanism for that construction was knowledge about how

[1] Miller's SPADE-0 [11] is another example of a programming tutor; unlike PROUST, it constrains the program construction process so that less machinery is required for understanding and more effort can be devoted to pedagogy.

programs are typically constructed, together with knowledge about novice misconceptions.

```
Problem:  Read in numbers, taking their sum, until the number 99999 is seen.  Report
the average.  Do not include the final 99999 in the average.

 1     PROGRAM Average( input, output );
 2     VAR Sum, Count, New, Avg: REAL;
 3     BEGIN
 4       Sum := 0;
 5       Count := 0;
 6       Read( New );
 7       WHILE New<>99999 DO
 8         BEGIN
 9           Sum := Sum+New;
10           Count := Count+1;
11           New := New+1
12         END;
13       Avg := Sum/Count;
14       Writeln( 'The average is ', avg );
15     END;
```

PROUST output:

```
It appears that you were trying to use line 11 to read the next input value.  Incrementing NEW
will not cause the next value to be read in.  You need to use a READ statement here, such as you
use in line 6.
```

Figure 1-1: Example of analysis of a buggy program

While we have not built a pedagogical expert yet, it would certainly need the type of information produced in the above analysis. That is, an intelligent tutoring system would need to know:

- what the bugs in the student's program are, and where they occur;
- what the student was intending to do with the buggy code;
- what misconceptions the student might have which would explain the presence of the bugs.

What is an appropriate method for deriving information such as this from a program? One way might be to compare the input-output behavior of the program against the expected input-output behavior. The information which this approach would provide is insufficient, particularly with larger programs, because a number of bugs might result in the same input/output behavior.[2]

---

[2]BIP [21] makes use of input/output behavior in its program analysis; consequently it only deals with small programming problems.

For example, many different bugs can cause a program to go into an infinite loop, so simply knowing that a program goes into an infinite loop is insufficient for determining what the bug is. Enhancing input-output analysis with dataflow analysis, or other compiler analysis techniques, will not help in cases where the code does not have any obvious structural anomalies,[3] such as in the preceding example.

What is missing in the above methods is a *detailed understanding of the relationship between the program text and the program's intentions.* We suggest that a method for building such a description involves (1) recreating the goals that the student was attempting to solve (i.e., what problem the student thought he was solving), (2) identifying the functional units in the program that were intended to realize those goals. In effect, *the programming expert needs to analyze the buggy program by reconstructing the manner in which it was generated.* The claim is that the trace generated by the programming expert does actually correspond to what the student was thinking, although not necessarily to the utmost detail; the pedagogical expert would then use that trace in subsequent tutoring activity.[4] In this paper, we briefly highlight the theoretical basis for reconstructive program analysis, and we detail how PROUST goes about building the reconstruction.

2. The Role of Plans in Program Understanding

Knowledge about what implementation methods should be used in programming is codified in PROUST in the form of *programming plans.* A programming plan is a procedure or strategy for realizing intentions in code, where the key elements have been abstracted and represented explicitly. It is our position that expert programmers make extensive use of programming plans, rather than each time building programs out of the primitive constructs of a programming language. This claim is based on a theory of what mental representations programmers have and use in reading and writing programs. In [17, 6, 19, 20] we describe various empirical experiments which support our theory. Thus, PROUST is directly based on a plausible, psychological theory of the programming process. Note that codifying programming knowledge in terms of plans is not unique to PROUST; the Programmer's Apprentice, [12] for example, also makes extensive use of plans.[5]

Figure 2-1 is an illustration of how plans are realized in programs. The figure shows a correct

---

[3]One area in which many compilers do a reasonable job is analyzing *syntactic* errors. Although it would be worthwhile to construct a parser which produces error reports aimed at novices, this is outside of the scope of our current work.

[4]Most intelligent tutoring systems at least tacitly assume such a correspondence. [7, 8, 3]

[5]Sniffer [15] is a prototype of a debugging system which is based upon the Programmer's Apprentice.

implementation of the problem shown in Figure 1-1, together with four plans that this program uses. Two of them, the RUNNING TOTAL VARIABLE PLAN and the COUNTER VARIABLE PLAN, are *variable plans*, i.e. they are plans which generate a result which is usually stored in a variable. Such plans typically have an initialization section and an update section, and carry information about what context they must appear in, e.g. whether or not they must be enclosed in a loop. The other two plans, the RUNNING TOTAL LOOP PLAN and the VALID RESULT SKIP GUARD, are *control plans*; their main role is not to generate results but to regulate the generation and use of data by other plans. The RUNNING TOTAL LOOP PLAN is a method for constructing a loop which controls the computation of a running total; in this program it also controls the operation of the COUNTER VARIABLE PLAN. The VALID RESULT SKIP GUARD plan is an example of a skip guard, i.e. a control plan which causes control flow to skip around other code when boundary conditions occur. In this case it prevents the average from being computed or output when there is no input.

```
Problem:  Read in numbers, taking their sum, until the number 99999 is seen.  Report
the average.  Do not include the final 99999 in the average.

                    PROGRAM Average( INPUT, OUTPUT );
                    VAR Sum, Count, New, Avg: REAL;
Counter Variable    BEGIN
Plan        ----------> Count := 0;
            |      ---> Sum := 0;                  Running Total Loop Plan
            |      |    Read(New); <-------------------
Running Total|     |    WHILE New <> 99999 DO <-------|
Variable Plan|     |      BEGIN                       |
            |      -------> Sum := Sum + New; <-------|
            --------------> Count := Count + 1;       |
                              Read(New);       <--------
                            END;                  Valid Result Skip Guard
                          IF Count > 0 THEN  <------------------------
                              BEGIN <--------------------------------|
                                Avg := Sum/Count; <------------------|
                                Writeln( Avg); <---------------------|
                              END <----------------------------------|
                            ELSE  <----------------------------------|
                              Writeln( 'no legal inputs'); <---------|
                        END.
```

Figure 2-1: Programming Plans

Recognition of plans in programs forms the basis of our approach to program understanding. But plan recognition alone is insufficient. Novices often use plans that would never occur to an expert, because they do not have a good sense of what is a good plan and what is not. PROUST's knowledge base of plans has therefore been extended in order to include many stylistically dubious plans.[6] Unfortunately, the more alternative plans there are in the system, the harder it

[6]The process of collecting novice programs and analyzing them is described in [2], [9], and [10].

is to determine which plans the programmer was using. Furthermore, program behavior depends not only upon what plans are used, but how they are organized; it is thus possible for a program to use correct plans yet still have bugs. In order to cope with these problems a method is needed for relating plans to other plans, and to the programmer's underlying intentions. This process, and the way it is used to search for the right interpretation of the program, is described in Section 4.

3. A Typical Problem in PROUST's Domain

PROUST's knowledge base is currently tailored to analyze the programming problem in Figure 3-1.[7] This problem (hereafter referred to as the Rainfall Problem) is a more complex version of the averaging problem shown in Figure 1-1. Among other computations, a program that solves this problem must

1. count the number of valid inputs (i.e., days on which there was zero or greater rainfall), and
2. count the number of positive inputs (i.e., days on which rain fell).

Novices attempt to realize these two goals in a variety of correct and buggy ways. Since coping with variability is one of PROUST's main objectives, examining how PROUST handles this specific set of goals should be illustrative. Thus, in what follows, we will focus on PROUST'S techniques for processing fragments of code that implement these goals.

> Noah needs to keep track of rainfall in the New Haven area in order to determine when to launch his ark. Write a program which he can use to do this. Your program should read the rainfall for each day, stopping when Noah types "99999", which is not a data value, but a sentinel indicating the end of input. If the user types in a negative value the program should reject it, since negative rainfall is not possible. Your program should print out the number of valid days typed in, the number of rainy days, the average rainfall per day over the period, and the maximum amount of rainfall that fell on any one day.

Figure 3-1: The Rainfall Problem

4. Relating Goals to Code via Plans

In order to relate the plans in a program to the program requirements, PROUST makes explicit the *goal decomposition* underlying the program. A goal decomposition consists of

- a description of the hierarchical organization of the subtasks in a problem,
- indications of the relationships and interactions among subtasks, and
- a mapping from subtask requirements (goals) to the plans that are used to implement them.

---

[7] We are currently extending PROUST to handle a range of introductory programming problems.

The plans which a goal decomposition specifies are matched against the program; this results in a mapping from program requirements to individual statements.

In attempting to understand all except the most trivial programming problems, two issues must be squarely faced:

- the goal decomposition of a *problem* may not be unique, and
- one *program* may be associated with more than one goal decomposition.

We deal with each issue in turn in the next two sections.

4.1. The Space of Goal Decompositions and Programs

Figure 4-1 illustrates how alternative goal decompositions can lead to different program implementations. A single problem description, at the top, can result in several different goal decompositions, which in turn result in a number of different programs, depending upon which plans are used. Some of these programs may be correct, others buggy. Buggy programs are either derived from incorrect goal decompositions or from incorrect implementations of correct goal decompositions. Each path from the problem description down to an individual program is a *program interpretation*; we call this set of possible derivation paths the *interpretation space* associated with a problem.

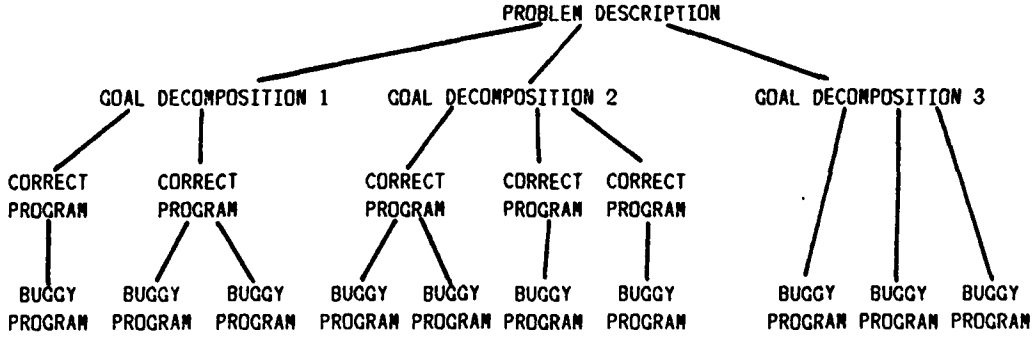


Figure 4-1: Search space of possible programs

Figures 4-2 and 4-3 illustrate two different solutions of the Rainfall Problem (Figure 3-1) and their corresponding goal decompositions. We focus here on two specific aspects of the problem:[8] (1) counting the valid inputs (daily rainfall greater than or equal to zero), and (2) counting the number of rainy days (daily rainfall strictly greater than zero).

Figure 4-2 shows a fragment in which these two goals are realized directly. First, a COUNTER VARIABLE PLAN is used to count the valid inputs; this is realized in the code that computes the

[8]There are other differences in the goal decompositions of these programs besides the ones mentioned here. However, we will not analyze them in this discussion.

value of the variable Valid. Second, the GUARDED COUNTER VARIABLE PLAN is used for counting the positive inputs; the variable Rainy is used in this plan.

While the program in Figure 4-3 also prints out the number of valid inputs and the number of positive inputs, the goal decomposition in this program is different. Instead of the two goals of counting the valid inputs and counting the positive inputs, the program in Figure 4-3 uses three goals to achieve the same end: (1) count the zero inputs, (2) count the positive inputs, and (3) add these two counters together to derive the valid day total. The goal of counting the positive inputs is implemented with a GUARDED COUNTER VARIABLE PLAN, operating on the variable Rainy. The goal of counting the zero inputs is also implemented with a GUARDED COUNTER VARIABLE PLAN, operating on the variable Dry. The counters are combined with an ADD PARTIAL RESULTS PLAN, resulting in the variable Valid.

4.2. Resolving Ambiguous Interpretations

If the mapping from problem descriptions to programs is to be rich enough to generate a sufficiently wide variety of programs, ambiguity is an unavoidable consequence, i.e. two different paths in the interpretation space can lead to the same program. This situation is exacerbated when buggy programs are allowed: bugs add uncertainty to the analysis. For example, if one encounters a statement New := New+1 in a correct program, one can be fairly certain that it is part of a counter plan. But if the program is buggy, as in Figure 1-1, one must also consider the possibility that this statement is intended to input new values; the only way of determining which is the proper role is by looking at the program as a whole and determining which interpretation is more consistent with the interpretations of the other parts of the program. The ability to enumerate and evaluate alternative interpretations is a key processing technique for a system that attempts to understand buggy programs.

In Figure 4-4 we give an example of the results of PROUST's attempt to resolve ambiguous interpretations. Figure 4-4 shows a fragment of code which might appear in a novice solution to the Rainfall Problem in Figure 3-1. We have focused on the counter variables in the program, Valid and Rainy; the rest of the main loop of the program is shown so that the surrounding context may be seen. Instead of counting the positive inputs (Rain>0) and the valid inputs (Rain>=0), this program counts the positive inputs and the zero inputs, and does not count the valid inputs.

There are two possible interpretations for this code, each of which results in a different explanation for the bugs. According to one interpretation, shown on the left side of the figure, the programmer intended to implement the valid input goal and the positive input goal directly. The plans used are COUNTER VARIABLE PLAN and GUARDED COUNTER VARIABLE PLAN; the resulting variables are Valid and Rainy, respectively. Valid appears to count only the the zero

Plans	Goal Decomposition
Plan: RUNNING TOTAL LOOP PLAN	1. *Get input, stopping at 99999* 2. *Check that input is non-negative* ...
Plan: COUNTER VARIABLE PLAN	6. *Count valid inputs*
Plan: GUARDED COUNTER VARIABLE PLAN	7. *Count positive inputs* ...

```
PROGRAM Rain1 (INPUT, OUTPUT);
  CONST STOP=99999;
  VAR
    Sum,Rain,Max,Ave: REAL;
    Valid,Rainy: INTEGER;
  BEGIN
    Writeln('Enter rainfall');
    Sum:=0;
    Valid:=0;
    Rainy:=0;
    Max:=0;
    Readln;
    Read(Rain);
    WHILE Rain<>STOP DO
      BEGIN
        IF Rain<0 THEN
          Writeln(Rain:0:2,' is not a possible rainfall, try again')
        ELSE
          BEGIN                          COUNTER VARIABLE PLAN <-- Count valid inputs
            Sum:=Sum+Rain;
            Valid:=Valid+1;
            IF Rain>Max THEN
              Max:=N;                    GUARDED COUNTER VARIABLE PLAN <-- Count positive inputs
            IF Rain>0 THEN
              Rainy:=Rainy+1;
          END;
        Writeln('Enter rainfall');
        Readln;
        Read(Rain)
      END;
    Writeln;
    Writeln(Valid:0,' valid rainfalls were entered.');
    IF Valid>0 THEN
      BEGIN
        Ave:=Sum/Valid;
        Writeln('The average rainfall was ',Ave:0:2,' inches PER DAY.');
        Writeln('The highest rainfall was ',Max:0:2,' Inches.');
        WRITELN('There were ',Rainy:0,' rainy days in this period. ')
      END
  END.
```

Figure 4-2: Simple goal decomposition

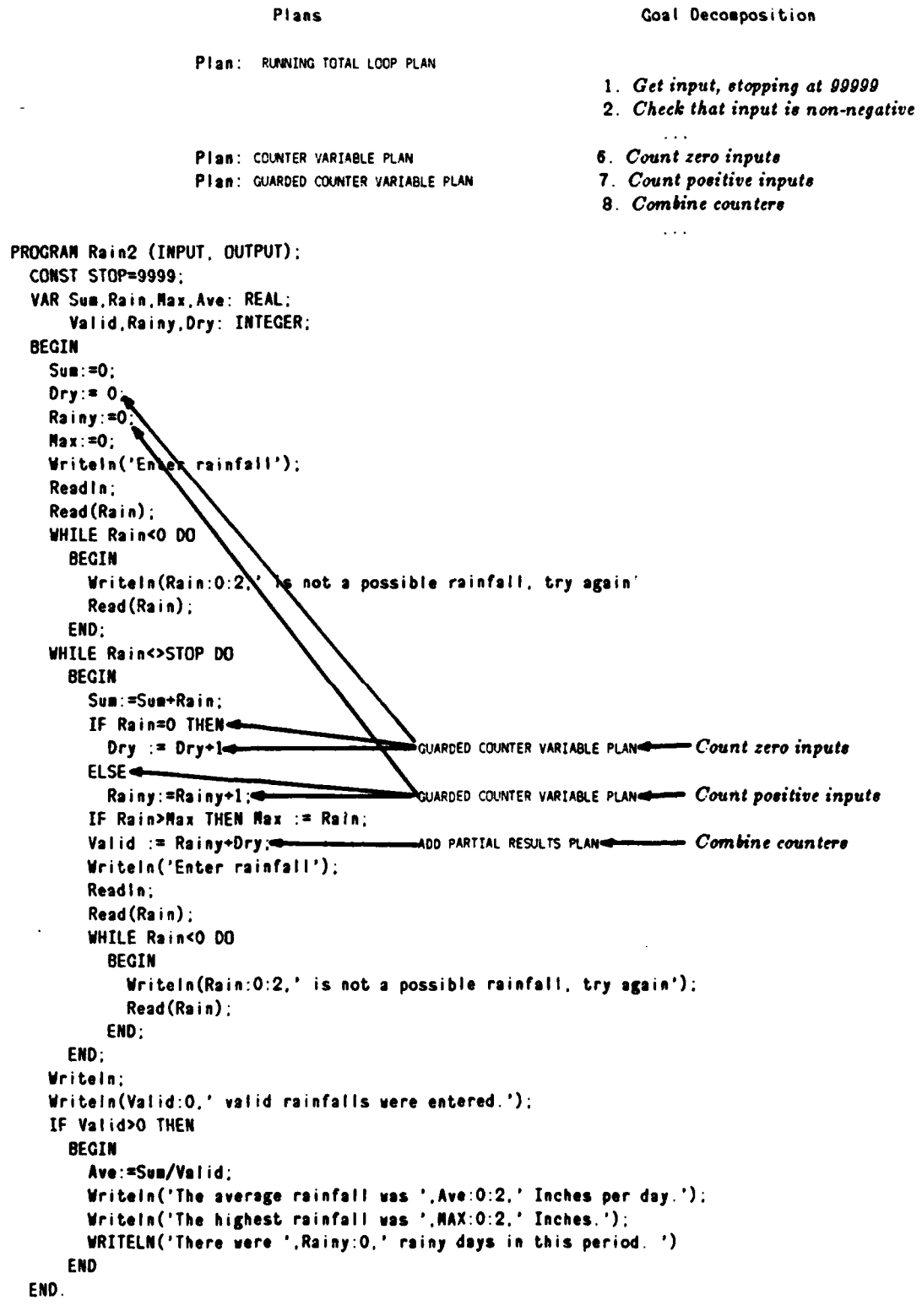


Figure 4-3: Alternative goal decomposition

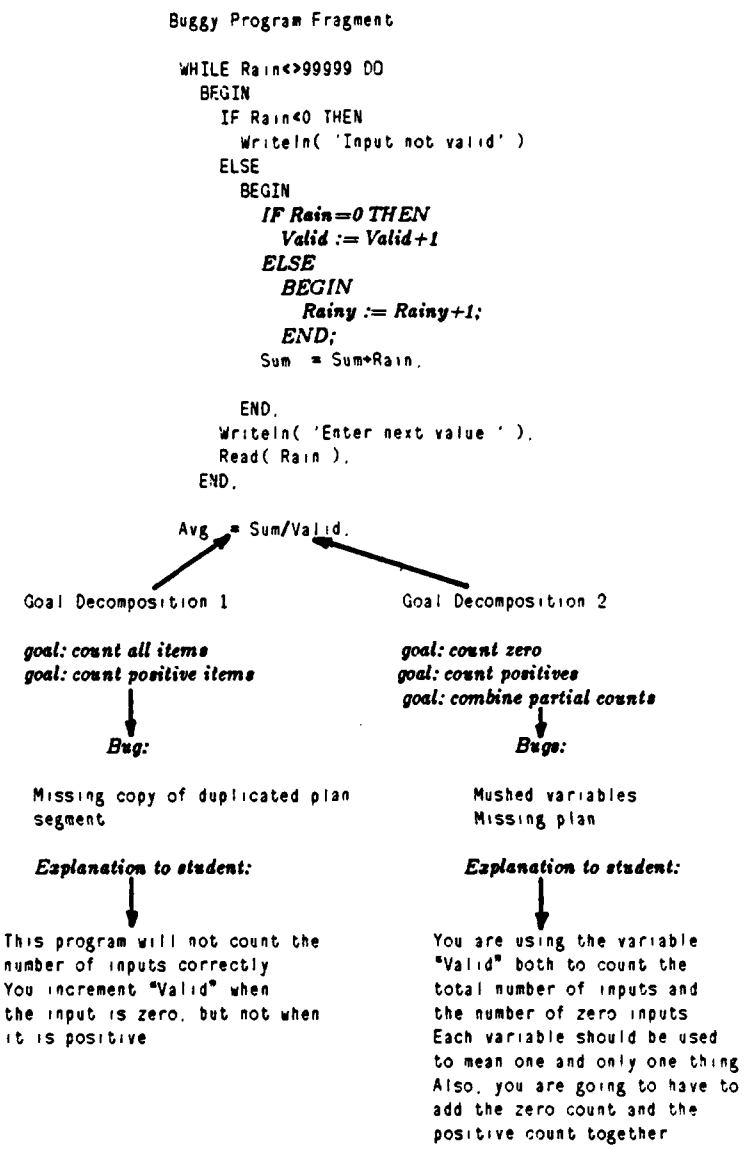


Figure 4-4: Alternative explananations for bugs

inputs, because the programmer intended to *modify* the COUNTER VARIABLE PLAN so that a copy of the counter update appears in both the THEN branch and the ELSE branch of the inner IF statement, and then left out one of the copies. The failure to update `Valid` in both branches thus appears to be a low-level slip, such as a mistake in editing the source file.

In the other interpretation, on the right side of the of the figure, the program is assumed to arise from a goal decomposition where the positive values and the zero values are counted separately and then added together. The programmer uses the variable `Valid` to refer to the count of zero values and `Rainy` to refer to the count of positive values. The plan to add `Valid` and `Rainy` together is missing. We could claim that the plan is missing because of an editing slip.

However, the context in which the counter plans appear weighs against this hypothesis: the average computation uses Valid in the denominator of the division, implying that Valid is the valid input counter as well as the zero input counter. We call variables which are used in contradictory ways such as this *mushed variables*. Mushed variables are very serious bugs; they reflect radical deficiencies in the programmer's ability to design programs. Therefore this goal decomposition is less highly valued than the previous goal decomposition. PROUST has a number of heuristics for deciding among alternative interpretations such as these.

5. The Understanding Process: An Example Of PROUST In Action

In the preceding sections, we (1) described what difficulties a program understanding system must overcome in order to analyze a program accurately, and we (2) gave an example of the *results* of PROUST's analysis. In this section, we will illustrate PROUST's *processing* capabilities. First we will describe the overall strategy by which PROUST searches through the space of potential interpretations for one that best accounts for the student's program, and then we will describe how PROUST actually produces the analysis already depicted in Figure 4-4.

5.1. Searching the Interpretation Space

Clearly, one can't possibly enumerate beforehand the space of program interpretations: there are just too many ways to construct correct and buggy programs. Rather, starting with the problem specification and a database of correct and buggy plans, transformation rules[9], and bug-misconception rules, PROUST constructs and evaluates interpretations for the program under consideration. In effect, the goal decomposition and the plan analysis of the program evolve simultaneously. To constrain the generation process, PROUST employs heuristics about what plans and goals are likely to occur together.

The evaluation process is *prediction driven:* based on the current candidate interpretation for the program, how well do *other parts of the program conform to PROUST's expectations*? For example, if, in a program that attempts to solve the Rainfall Problem, PROUST has assumed that the variable Count is keeping track of the number of valid days, PROUST would expect to see Count in the denominator of the average daily rainfall calculation. If this expectation is confirmed, then PROUST is more confident of its interpretation, and vice versa. PROUST employs heuristics that evaluate matches, near-misses, and misses of its expectations. Examples of construction and evaluation processes will be given in the next section.

The fact that PROUST constructs and evaluates interpretations anew for each program, and

---

[9]These entities will be explained shortly.

does not rely on a prestored set of possible interpretations, provides it with an important capability: PROUST readily generates interpretations for programs that it (and we) have not seen previously. That is, unlike some diagnostic systems that effectively choose a fault from a set of predefined faults, [16, 4] PROUST actively constructs diagnoses. Given the variability in programs, PROUST needs such a capability in order to be effective.[10]

5.2. Putting It All Together: Two Examples

```
Sum := 0;
Rainy := 0;
Valid := 0;
Max := 0;
Read( Rain );
WHILE Rain<>99999 DO
  BEGIN
    IF Rain<0 THEN
        Writeln( 'Input not valid' )
    ELSE
      BEGIN
        IF Rain=0 THEN                (a)
          Valid := Valid+1            (b)
        ELSE
          BEGIN
            Valid := Valid+1;         (c)
            Rainy := Rainy+1;
          END;
        Sum := Sum+Rain;
        IF Rain>Max THEN
            Max := Rain;
      END;
    Writeln( 'Enter next value:' );
    Read( Rain );
  END;
Avg := Sum/Valid;
```

Figure 5-1: Excerpt of Rainfall Program

In this section we will illustrate how PROUST actually goes about analyzing a program. We will show two examples; one is a correct program and the other is a buggy program.

5.2.1. Analysis of a correct program

Our first example, in Figure 5-1, is an excerpt from a correct solution to the Rainfall Problem in Figure 3-1; it is based on the program fragment shown in Figure 4-4. Although this program functions correctly, there is one construction which is unusual; the valid input counter `Valid` is updated in two places instead of one. That is, `Valid` is updated in each branch of the conditional

---

[10] FALOSY [14] is also capable for recognizing novel faults; however, it assumes that there is only one fault, which the programmer must describe beforehand.

statement at (a); the update at (b) is executed when Rain is zero, and the update at (c) when Rain is positive. The program in this figure illustrates the variability possible in programs; coping with this type of situation requires additional machinery, as will be seen shortly.

Assume that PROUST has carried out a partial plan analysis of this program already, and has made the following tentative assumptions:

- the variable Sum is the running total variable,
- the variable Valid keeps tracks of the number of valid days,
- the update on Valid should be in the loop, embedded inside a test for negative rainfall (IF Rain < 0 THEN....).

The processing that continues from this point is illustrated in Figure 5-2. PROUST maintains an agenda of goals that remain to be worked on; at this point in the analysis the agenda includes the Count goal for valid inputs, the Sum goal, and the Count goal for positive inputs, to name a few. PROUST selects the first goal on the agenda, as shown at (a), checks that it is ready for analysis, and then determines whether or not it needs to be decomposed. The entry in the knowledge base for Count stipulates that it is most commonly implemented in an undecomposed fashion, so Proust consults the plan database looking for appropriate plans for realizing this goal. It finds only one plan plan: the COUNTER VARIABLE PLAN (b). It then makes tentative bindings for the plan variables, and determines where each segment of the plan should be found. The resulting structure, shown at (c), can then be matched against the student's program.

Figure 5-3 shows the results of matching the instantiated plan against the code. There is a unique match for the initialization step of the plan, but instead of there being one match for the update step, there are *two* matches. Furthermore, PROUST expects the update to be at "top level" inside the loop, i.e. it should not be enclosed inside code which might disrupt its function. Instead it discovers that each update is enclosed in an IF statement which restricts its application. PROUST treats the plan as a near-match for the program, but the plan cannot be accepted until the match discrepancies are accounted for.

PROUST has a number of different methods for explaining a plan difference; one of them is to use *transformation rules* to relate the code to the plan. One such transformation is shown in Figure 5-4.[11] Each transformation rule has a test part and an action part. The test part consists of a conjunction of micro-tests, each testing various aspects of the program; the action part usually indicates how to modify the code in order to nullify the effect of the transformation. In this case the Distribution Transformation Rule applies. This is a rule for recognizing plans in situations where a set of computations have been divided into parts using a CASE statement or an

[11]PROUST currently has 15 such transformations in its database. Some rules, such as the Distribution Transformation Rule, are quite general; others, such as the transformation which changes Valid<>0 into Valid>0 if Valid is a counter variable, are plan specific.

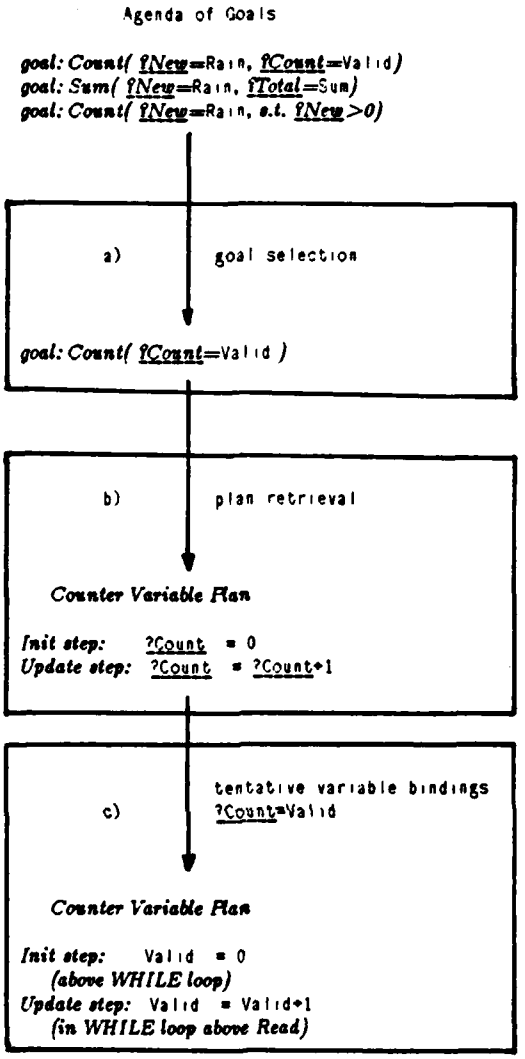


Figure 5-2: Simple mapping from goals to instantiated plans

IF-THEN-ELSE construct, and where the plan update is duplicated so that a copy appears in each branch. The control flow branches in this case are the two branches in the IF-THEN-ELSE construction which test for Rain=0 and Rain>0. The rule checks to see whether there is exactly one Valid:=Valid+1 statement for each possible branch of the test. It then combines the two updates and moves the result to an appropriate place outside of the test. Once this is done the counter plan matches successfully.

5.2.2. A buggy example

We will now show how PROUST analyzes the buggy program shown in Figure 4-4; a more complete version is given in Figure 5-5. When PROUST analyzes buggy programs such as this, it goes through much the same process that it goes through in analyzing correct programs; the main difference is that PROUST must consider more alternative interpretations in order to find the most

```
Sum := 0;
Valid := 0;                                   Valid := 0    Init step:
Rainy := 0;                                   EXACT MATCH
Max := 0;
Read( Rain );
WHILE Rain<>99999 DO
  BEGIN
    IF Rain<0 THEN
        Writeln( 'Input not valid' )
    ELSE
      BEGIN
        IF Rain=0 THEN
          Valid := Valid+1                    Valid := Valid+1    Update step:
        ELSE                                  TWO MATCHES; BOTH EMBEDDED INSIDE
          BEGIN                               UNEXPECTED CODE
            Valid := Valid+1;
            Rainy := Rainy+1;
          END;
        Sum := Sum+Rain;
        IF Rain>Max THEN
          Max := Rain;
      END;
    Writeln( 'Enter next value:' );
    Read( Rain );
  END;
Avg := Sum/Valid;
```

Figure 5-3: Plan matching

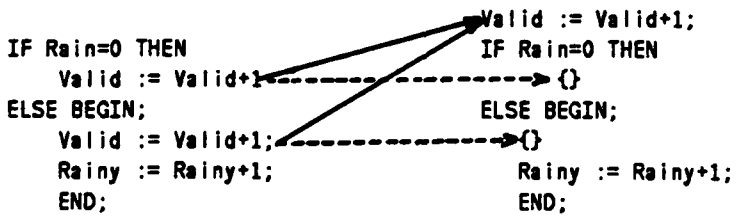


Figure 5-4: Program transformation

plausible explanation for the bug.

Figure 5-6 shows what happens when the COUNTER VARIABLE PLAN is matched against this program. This time there is one good match for the counter update; unfortunately it is inside of an unexpected IF statement. The Distribution Transformation Rule is invoked to explain the plan difference, but it predicts that there should be two updates, so it does not fully explain the problem. PROUST therefore looks for another rule which will explain the difference between the prediction made by the Distribution Transformation Rule and the observed code. A rule applies which states that if an single instance of duplicated code is missing, it is explainable as a low-level slip. This completes the mapping from the plan to the code.

Whenever an interpretation presumes the presence of a bug, it is necessary to make sure that

```
Sum := 0;
Rainy := 0;
Valid := 0;
Max := 0;
Read( Rain );
WHILE Rain<>99999 DO
  BEGIN
    IF Rain<0 THEN
        Writeln( 'Input not valid' )
    ELSE
      BEGIN
        IF Rain=0 THEN
          Valid := Valid+1
        ELSE
          BEGIN
            Rainy := Rainy+1;
          END;
        Sum := Sum+Rain;
        IF Rain>Max THEN
            Max := Rain;
      END;
    Writeln( 'Enter next value:' );
    Read( Rain );
  END;
Avg := Sum/Valid;
```

Figure 5-5: A buggy program

there are no other interpretations which presume fewer or less severe bugs. PROUST therefore goes back and looks for another way of of implementing the *Count* goal. PROUST has in its knowledge base an alternative method for decomposing *Count* goals, namely to implement counters for particular intervals and then combine the partial counts. One of these subgoals can be unified with the *Count positives* goal that already exists in the agenda. The two *Count* goals are thus transformed into a set of three goals. Plans can then be chosen and instantiated for each of these goals, as was done in Figure 5-2. The result plans, and the results of matching them, is shown in Figure 5-7. This time two match errors are found. First, `Valid` is the counter for zero values; but the average predicts that `Valid` is the main counter; `Valid` is a mushed variable. Second, the ADD PARTIAL RESULTS PLAN is missing altogether. PROUST ranks bugs according to their severity; missing plans that do not pertain to some boundary condition are moderately severe bugs, and mushed variables are extremely severe bugs. Therefore this interpretation is less highly valued, and the analysis involving the transformation holds.

```
Sum := 0;
Valid := 0;<-------------------------- Valid := 0  Init step:
Rainy := 0;                            EXACT MATCH
Max := 0;
Read( Rain );
WHILE Rain<>99999 DO
  BEGIN
    IF Rain<0 THEN
        Writeln( 'Input not valid' )
    ELSE
      BEGIN
        IF Rain=0 THEN
          Valid := Valid+1<------------ Valid := Valid+1  Update step:
        ELSE                           predicted by distribution transformation
          BEGIN
            Rainy := Rainy+1;          Valid := Valid+1 Update step:
          END;                         ??? condition for transformation violated
        Sum := Sum+Rain;                 EXPLANATION: low-level slip
        IF Rain>Max THEN
            Max := Rain;
      END;
    Writeln( 'Enter next value:' );
    Read( Rain );
  END;
Avg := Sum/Valid;
```

Figure 5-6: Transformation with bugs

6. Performance -- Preliminary Results

As a preliminary test of PROUST's capabilities, we tested PROUST on 206 different novice solutions to the Rainfall Problem shown above. We collected these programs by modifying the Pascal compiler used by the students in an introductory programming course so that each syntactically correct version of the program was stored on tape [2]. We ran PROUST on the first syntactically correct version from each student, so that we could see how PROUST behaves when faced with a large number of bugs.

In Table 6-1 we see how PROUST performed on this corpus of programs. Of the 206 programs in the sample, PROUST only commented on 137 of them (67%). The remaining 33% PROUST decided that it didn't understand the program well enough to make a reasonable assessment of the bugs. Thus, rather than venturing a guess, PROUST remained silent. On those programs that it did feel confident of its analysis, it was correct almost[12] 94% of the time! In an educational setting, we felt that no advice was better than bad advice. Thus, we built into

[12]There were still 32 "false alarms:" cases where PROUST said there was a bug, but there really wasn't.

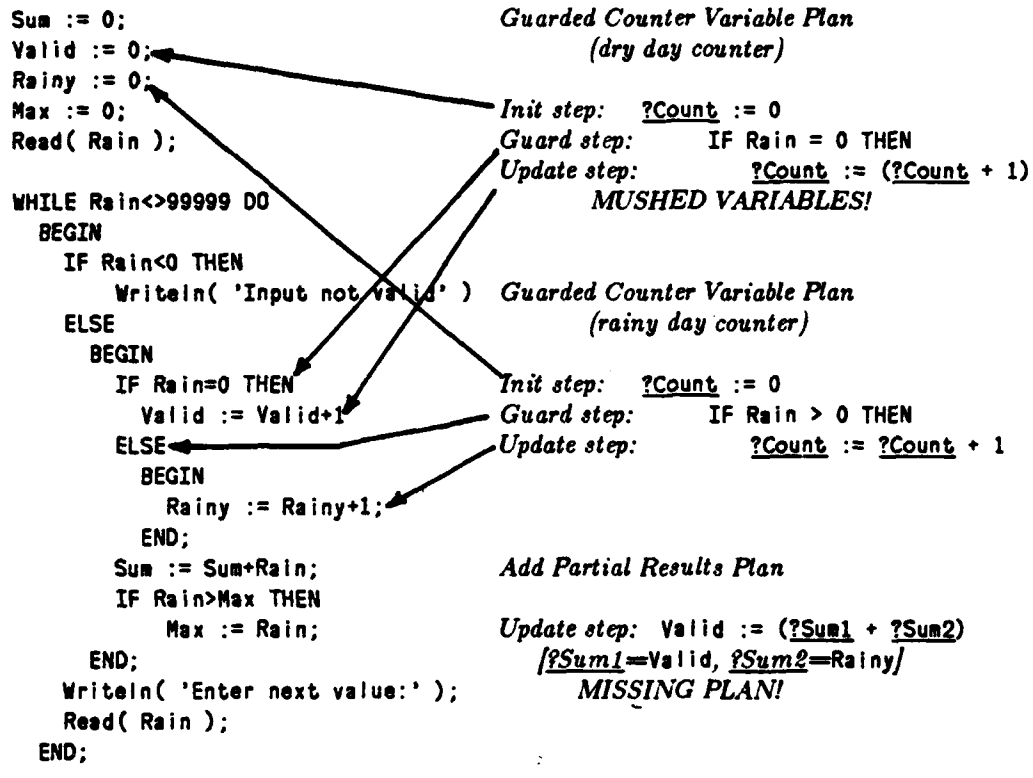


Figure 5-7: Matching alternate plans

PROUST a number of heuristics that it would use to assess its confidence in its analysis. From the data in Table 6-1, it seems that when PROUST thought it had a good analysis, it was indeed correct.

Total number of programs:	206
PROUST actually gave complete bug reports for 137 programs (67%)	
Total number of bugs (from 137 programs)	444
Bugs recognized correctly:	419 (94%)
Bugs not reported:	25 (6%)
False alarms:	32

Table 6-1: Preliminary results

Clearly, the next stage is to improve PROUST's overall performance. Moreover, in looking at the cases where PROUST failed, we see no fundamental obstacle to getting PROUST up to the 80% overall correct rate. However, we can can characterize the kinds of programs which will *always* cause problems for PROUST as follows: 1) very unusual bugs, which occur too infrequently to justify inclusion in PROUST's knowledge base, 2) programs which contain novel plans which PROUST has no means for predicting, 3) ambiguous cases which can only be resolved through dialog with the student. For these cases, we would suggest that the student see

a human teacher.

7. Concluding Remarks

Is all the machinery described in this paper necessary in order to understand buggy and correct programs -- programs that are only about 1 page in length? The answer, in our minds at least, is: undeniably yes. If anything, PROUST is the minimum that is required! The basis for this conclusion is twofold:

1. In Artificial Intelligence research, systems have been built to understand stories of moderate length that require machinery similar to that employed by PROUST. [13, 5] Certainly, programs are as complicated an entity as are stories.
2. We attempted to build a bug finding system that used a database of bug templates in a *context-independent* fashion to analyze programs similar to those analyzed by PROUST. That system, MENO, [16] failed miserably: in order to cope with the variety and variability in actual programs, a system must be able to understand how the pieces of the program fit together -- which is a highly *context-dependent* process.

Finally, all programmers intuitively know that the mapping from problem specifications to code is a complex process. What PROUST has done -- which we feel is its major contribution -- is lay that mapping process open to inspection: since PROUST constructs a program in its attempt to understand the program under analysis, we can "see" the programming process in action. By making the programming process explicit, our work joins with that of the software engineering community to change programming from an ethereal art to an object of scientific inquiry.

I. Proust's analysis of a sample program

```
 1  PROGRAM RAINFALL   (INPUT ,OUTPUT);
 2
 3  CONST
 4    SENTINEL = 99999;
 5
 6  VAR
 7    RAINFALL, VALID, HIGHEST, AVERAGE, TOTAL : REAL;
 8    RAINDAY : INTEGER;
 9 BEGIN
10 (*INFORMATION IS ENTERED INTO THE TERMINAL*)
11    WRITELN('PLEASE ENTER THE AMOUNT OF RAINFALL FOR EACH DAY SEPERATLY');
12    WRITELN('THIS PROGRAM WILL THEN FIGURE OUT THE AVERAGE, HIGHEST, TOTAL');
13    WRITELN('NUMBER OF RAINY DAYS, AND THE NUMBER OF VALID RAINY DAYS ENTERED');
14    WRITELN('PLEASE MAKE SURE THE NUMBERS ARE POSITIVE');
15    WRITELN('ENTER RAINFALL');
16    READLN;
17    READ(RAINFALL);
18
19    (*TEST FOR INVALID ENTRY*)
20      IF RAINFALL < 0 THEN
21    WRITELN('THE DATA IS IMPOSSIBLE PLEASE CHECK AND REENTER DATA');
22    WRITELN('PLEASE REMEMBER THE NUMBERS MUST BE POSITIVE');
23    READLN;
24    READ(RAINFALL);
25
26    (*IDENTIFICATION AND CALCULATIONS*)
27
28    BEGIN
29     WHILE RAINFALL <> SENTINEL DO
30     IF RAINFALL > HIGHEST THEN
31      HIGHEST := RAINFALL;
32      TOTAL := RAINFALL + TOTAL;
33      IF RAINFALL >= 0 THEN
34      VALID := VALID + 1;
35     IF RAINFALL > 0 THEN
36      RAINDAY := RAINDAY + 1;
37   END;
38 REPEAT
39      UNTIL RAINFALL = SENTINEL;
40
41    (*FIND AVERAGE*)
42    AVERAGE := (TOTAL / VALID);
43
44     (*PRINTS OUTPUT*)
45    WRITELN(' ',VALID:2,'VALID RAINFALLS WERE ENTERED.');
46    WRITELN('THE AVERAGE RAINFALL WAS',AVERAGE:8:2,'INCHES PER DAY');
47    WRITELN('THE HIGHEST RAINFALL WAS',HIGHEST:8:2,'INCHES');
48    WRITELN('THERE WERE',RAINDAY:2,'RAINY DAYS IN THIS PERIOD')
49  END.
```

Bug Report

1. The WHILE statement at line 29 and the BEGIN statement at line 29 are reversed. The BEGIN should come after the WHILE.

2. Your main loop is missing a READ statement. As it stands your loop will process the same input value over and over.
Perhaps you intended line 24 to serve this purpose. If so, there may be a bug there.

3. You left out the initialization for the variable VALID.

4. You left out the initialization for the variable TOTAL.

5. You left out the initialization for the variable RAINDAY.

6. You left out the initialization for the variable HIGHEST.

7. You need a test to check that at least one valid data point has been input before line 42 is executed. The average is not defined when there is no input.

8. You need a test to check that at least one valid data point has been input before line 46 is executed. The average is not defined when there is no input.

9. You need a test to check that at least one valid data point has been input before line 47 is executed. The maximum is not defined when there is no input.
Perhaps you intended line 33 to serve this purpose. If so, there may be a bug there.

10. Your test for valid input at line 20 won't work, because it's outside the main loop. Remember that you have to test ALL the input for validity.

11. The loop at line 39 doesn't do anything; it will loop forever.
In particular, it will not make the program loop back to the beginning, if that is what you had in mind.

References

[1] Bonar, J. and Soloway, E.
Uncovering Principles of Novice Programming.
1983.
SIGPLAN-SIGACT Tenth Symposium on the Principles of Programming Languages, in press.

[2] Bonar, J., Ehrlich, K., Soloway, E.
Collecting and Analyzing On-Line Protocols from Novice Programmers .
Behavioral Research Methods and Instrumentation 14:203-209, 1982.

[3] Brown, J. S., Burton, R. R., and de Kleer, J.
Pedagogical, Natural Language and Knowledge Engineering Techniques in SOPHIE I, II. and III.
In Sleeman, D. and Brown, J. S. (editors), *Intelligent Tutoring Systems*. Academic Press, New York, 1981.

[4] Clancey, W. J., Bennett, J. S., and Cohen, P. R.
Applications-oriented AI Research: Education.
Technical Report HPP-79-17, Stanford Heuristic Programming Project, July, 1979.

[5] Dyer, M.
In-Depth Understanding.
Technical Report 219, Computer Science Department, Yale University, May, 1982.

[6] Ehrlich, K., Soloway, E.
An Empirical Investigation of the Tacit Plan Knowledge in Programming.
1983.
in Human Factors in Computer Systems , J. Thomas and M.L. Schneider (Eds.), Ablex Inc., in press.

[7] Genesereth, M. R.
The Role of Plans in Intellegent Teaching Systems.
In Brown, J. S. and Sleeman, D. (editors), *Intellegent Tutoring Systems*. New York, 1981.

[8] Goldstein, I. P.
The Genetic Graph: a Representation for the Evolution of Procedural Knowledge.
Int. J. of Man-Machine Studies 11:51-77, 1979.

[9] Johnson, L., Draper, S., and Soloway, E.
An Effective Bug Classification Scheme Must Take the Programmer into Account.
1983.
SIGPLAN/SIGSOFT Workshop on High-Level Debugging, in press.

[10] Johnson, L., Draper, S., and Soloway, E.
Classifying Bugs is a Tricky Business.
1983.
NASA Workshop on Software Engineering, in press.

[11] Miller, M. L.
A Structured Planning and Debugging Environment for Elementary Programming.
Int. J. of Man-Machine Studies 11:79-95, 1978.

[12] Rich, C.
A Formal Representation for Plans in the Programmer's Apprentice.
In *Proc. of the Seventh Int. Joint Conf. on Artificial Intelligence*, pages 1044-1052. ICJAI, August, 1981.

[13] Schank, R. and Abelson, R.
Scripts, Plans, Goals, and Understanding.
Lawrence Erlbaum, Hillsdale, New Jersey, 1977.

[14] Sedlmeyer, R. L. and Johnson, P. E.
Diagnostic Reasoning in Software Fault Localization.
In *Proceedings of the SIGSOFT Workshop on High-Level Debugging.* SIGSOFT, Asilomar, Calif., 1983.

[15] Shapiro, D. G.
Sniffer: a System that Understands Bugs.
Technical Report AI Memo 638, MIT Artificial Intelligence Laboratory, June, 1981.

[16] Soloway, E., Rubin. E., Woolf, B., and Bonar, J.
MENO-II: An AI-CAI Programming Tutor.
1983.
Journal of Computer-Based Instruction, in press.

[17] Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J.
What do Novices Know about Programming.
In A. Badre and B. Shneiderman (editor), *Directions in Human-Computer Interactions*. Ablex Inc., Norwood, New Jersey, 1982.

[18] Soloway, E., Ehrlich, K., Bonar, J.
Tapping Into Tacit Programming Knowledge.
In *Proceedings of the Conference on Human Factors in Computing Systems.* NBS, Gaithersburg, Md., 1982.

[19] Soloway, E., Ehrlich, K., and Gold, E.
Reading a Program Is Like Reading a Story (Well, Almost).
In *Proceedings of the Cognitive Science Conference, 1983.* Cognitive Science Society, Rochester, N.Y., 1983.

[20] Soloway, E., Bonar, J., and Ehrlich, K.
Cognitive Strategies and Looping Constructs: An Empirical Study.
1983.
Communications of the ACM, in press.

[21] Wescourt, K. T., Beard, M., Gould, L., and Barr, A.
Knowledge-based CAI: CINS for Individualized Curriculum Sequencing.
Technical Report 290, Stanford Institute for Mathematical Studies in the Social Sciences, Psychology and Education Series, October, 1977.

-- OFFICIAL DISTIRUBTION LIST --

Army

Technical Director
U S Army Research Institute for the Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, Virginia 22333
1 copy

Mr James Baker
Army Research Institute
5001 Eisenhower Avenue
Alexandria, Virginia 22333
1 copy

Dr Beatrice J Farr
U S Army Research Institute
5001 Eisenhower Avenue
Alexandria, Virginia 22333
1 copy

Dr Milton S Katz
Williams Technical Area
U S Army Research Institute
5001 Eisenhower Avenue
Alexandria, Virginia 22333
1 copy

Dr Marshall Narva
U S Army Research Institute for the Behavioral & Social Sciences
5001 Eisenhower Avenue
Alexandria, Virginia 22333
1 copy

Dr Harold F O'Neil, Jr
Director, Training Research Lab
Army Research Institute
5001 Eisenhower Avenue
Alexandria, Virginia 22333
1 copy

Commander, US Army Research Institute for the Behavioral & Social Sciences
Attn PERI-BR (Dr Judith Orasanu)
5001 Eisenhower Avenue
Alexandria, Virginia 22333
1 copy

Joseph Psotka, Ph D
Attn PERI-IC
Army Research Institute
5001 Eisenhower Avenue
Alexandria, Virginia 22333
1 copy

Dr Robert Sasmor
U S Army Research Institute for the Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, Virginia 22333
1 copy

Dr Robert Wisher
Army Research Institute
5001 Eisenhower Avenue
Alexandria, Virginia 22333
1 copy

Private Sector

Dr Michael Genesereth
Department of Computer Science
Stanford University
Stanford, California 94305
1 copy

Dr Dedre Gentner
Bolt Beranek & Newman
10 Moulton Street
Cambridge, Massachusetts 02138
1 copy

Dr Robert Glaser
Learning Research & Development Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, Pennsylvania 15260
1 copy

Dr Joseph Goguen
SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
1 copy

Dr Bert Green
Johns Hopkins University
Department of Psychology
Charles & 34th Street
Baltimore, Maryland 21218
1 copy

Dr James G Greeno
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, Pennsylvania 15213
1 copy

Dr Barbara Hayes-Roth
Department of Computer Science
Stanford University
Stanford, California 95305
1 copy

Dr Frederick Hayes-Roth
Teknowledge
525 University Avenue
Palo Alto, California 94301
1 copy

Glena Greenwald, Ed
Human Intelligence Newsletter
P O Box 1163
Birmingham, Michigan 48012
1 copy

Dr Earl Hunt
Department of Psychology
University of Washington
Seattle, Washington 98105
1 copy

Dr Marcel Just
Department of Psychology
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213
1 copy

Air Force

U S Air Force Office of Scientific Research
Life Sciences Directorate, NL
Bolling air Force Base
Washington, DC 20332
1 copy

Dr Earl A Alluisi
HQ AFHRL (AFSC)
Brooks AFB, Texas 78235
1 copy

Bryan Dallman
AFHRL/LRT
Lowry AFB, Colorado 80230
1 copy

Dr Genevieve Haddad
Program Manager
Life Sciences Directorate
AFOSR
Bolling AFB, DC 20332
1 copy

Dr John Tangney
AFOSR/NL
Bolling AFB, DC 20332
1 copy

Dr Joseph Yasutake
AFHRL/LRT
Lowry AFB, Colorado 80230
1 copy

Marine Corps

H William Greenup
Education Advisor (E031)
Education Center, MCDEC
Quantico, Virginia 22134
1 copy

Special Assistant for Marine Corps Matters
Code 100M
Office of Naval Research
800 N Quincy Street
Arlington, Virginia 22217
1 copy

Dr A L Slafkosky
Scientific Advisor (Code RD-1)
HQ, U S Marine Corps
Washington, DC 20380
1 copy

Department of Defense

Defense Technical Information Center
Cameron Station, Bldg 5
Alexandria, Virginia 22314
Attn TC
12 copies

Military Assistant for Training and Personnel Technology
Office of the Under Secretary of Defense for Research & Engineering
Room 3D129, The Pentagon
Washington, DC 20301
1 copy

Major Jack Thorpe
DARPA
1400 Wilson Blvd
Arlington, Virginia 22209
1 copy

Dr David Kieras
Department of Psychology
University of Arizona
Tucson, Arizona 85721
1 copy

Dr Walter Kintsch
Department of Psychology
University of Colorado
Boulder, Colorado 80302
1 copy

Dr Stephen Kosslyn
Department of Psychology
The John Hopkins University
Baltimore, Maryland 21218
1 copy

Dr Pat Langley
The Robotics Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213
1 copy

Dr Jill Larkin
Department of Psychology
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213
1 copy

Dr Alan Lesgold
Learning R&D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, Pennsylvania 15213
1 copy

Dr Jim Levin
University of California
at San Diego
Laboratory for Comparative Human Cognition - D003A
La Jolla, California 92093
1 copy

Dr Michael Levine
Department of Educational Psychology
210 Education Bldg
University of Illinois
Champaign, Illinois 61801
1 copy

Dr Marcia Linn
University of California
Director, Adolescent Reasoning Project
Berkeley, California 94720
1 copy

Dr Jay McClelland
Department of Psychology
MIT
Cambridge, Massachusetts 02139
1 copy

Dr James R Miller
Computer Thought Corporation
1721 West Plano Highway
Plano, Texas 75075
1 copy

Dr Mark Miller
Computer Thought Corporation
1721 West Plano Highway
Plano, Texas 75075
1 copy

Navy

Robert Ahlers — 1 copy
Code M711
Human Factors Laboratory
NAVTRAEQUIPCEN
Orlando, Florida 32813

Code M711 — 1 copy
Attn: Arthur S Blaiwes
Naval Training Equipment Center
Orlando, Florida 32813

Liaison Scientist — 1 copy
Office of Naval Research
Branch Office, London
Box 39
FPO New York, New York 09510

Dr Richard Cantone — 1 copy
Navy Research Laboratory
Code 7510
Washington, DC 20375

Chief of Naval Education and Training — 1 copy
Liason Office
Air Force Human Resource Laboratory
Operations Training Division
WILLIAMS AFB, Arizona 85224

Dr Stanley Collyer — 1 copy
Office of Naval Technology
800 N Quincy Street
Arlington, Virginia 22217

CDR Mike Curran — 1 copy
Office of Naval Research
800 N Quincy Street
Code 270
Arlington, Virginia 22217

Dr John Ford — 1 copy
Navy Personnel R&D Center
San Diego, California 92152

Dr Jude Franklin — 1 copy
Code 7510
Navy Research Laboratory
Washington, DC 20375

Dr Mike Gaynor — 1 copy
Navy Research Laboratory
Code 7510
Washington, DC 20375

Dr Jim Hollan — 1 copy
Code 14
Navy Personnel R&D Center
San Diego, California 92152

Dr Ed Hutchins — 1 copy
Navy Personnel R&D Center
San Diego, California 92152

Dr Tom Moran — 1 copy
Xerox PARC
3333 Coyote Hill Road
Palo Alto, California 94304

Dr Allen Munro — 1 copy
Behavioral Technology Laboratories
1845 Elena Avenue, Fourth Floor
Redondo Beach, California 90277

Dr Donald Norman — 1 copy
Cognitive Science, C-015
Univ of California, San Diego
La Jolla, California 92093

Dr Jesse Orlansky — 1 copy
Institute for Defense Analyses
1801 N Beauregard Street
Alexandria, Virginia 22311

Professor Seymour Papert — 1 copy
20C-109
MIT
Cambridge, Massachusetts 02139

Dr Nancy Pennington — 1 copy
University of Chicago
Graduate School of Business
1101 E 58th Street
Chicago, Illinois 60637

Dr Richard A Pollak — 1 copy
Director, Special Projects
MECC
2354 Hidden Valley Lane
Stillwater, Minnesota 55082

Dr Peter Polson — 1 copy
Department of Psychology
University of Colorado
Boulder, Colorado 80309

Dr Fred Reif — 1 copy
Physics Department
University of California
Berkeley, California 94720

Dr Lauren Resnick — 1 copy
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, Pennsylvania 15213

Mary S Riley — 1 copy
Program in Cognitive Science
Center for Human Information Processing
University of California, San Diego
La Jolla, California 92093

Dr Andrew Rose — 1 copy
American Institutes for Research
1055 Thomas Jefferson Street, NW
Washington, DC 20007

Dr Norman J Kerr — 1 copy
Chief of Naval Technical Training
Naval Air Station Memphis (75)
Millington, Tennessee 38054

Dr James Lester — 1 copy
ONR Detachment
495 Summer Street
Boston, Massachusetts 02210

Dr William L Maloy (02) — 1 copy
Chief of Naval Education and Training
Naval Air Station
Pensacola, Florida 32508

Dr Joe McLachlan — 1 copy
Navy Personnel R&D Center
San Diego, California 92152

Dr William Montague — 1 copy
NPRDC Code 13
San Diego, California 92152

Library, Code P201L — 1 copy
Navy Personnel R&D Center
San Diego, California 92152

Technical Director — 1 copy
Navy Personnel R&D Center
San Diego, California 92152

Commanding Officer — 6 copies
Naval Research Laboratory
Code 2627
Washington, DC 20390

Office of Naval Research — 1 copy
Code 433
800 N Quincy Street
Arlington, Virginia 22217

Personnel & Training Research Group — 6 copies
Code 442PT
Office of Naval Research
Arlington, Virginia 22217

Office of the Chief of Naval Operations — 1 copy
Research Development & Studies Branch
OP 115
Washington, DC 20350

LT Frank C Petho, MSC, USN (Ph D) — 1 copy
CNET (N-432)
NAS
Pensacola, Florida 32508

Dr Gary Poock — 1 copy
Operations Research Development
Code 55PK
Naval Postgraduate School
Monterey, California 93940

Dr Ernst Z Rothkopf — 1 copy
Bell Laboratories
Murray Hill, New Jersey 07974

Dr William B Rouse — 1 copy
Georgia Institute of Technology
School of Industrial & Systems
Engineering
Atlanta, Georgia 30332

Dr David Rumelhart — 1 copy
Center for Human Information Processing
University of California, San Diego
La Jolla, California 92093

Dr Michael J Samet — 1 copy
Perceptronics, Inc
6271 Variel Avenue
Woodland Hills, California 91364

Dr Roger Schank — 1 copy
Yale University
Department of Computer Science
P O Box 2158
New Haven, Connecticut 06520

Dr Walter Schneider — 1 copy
Psychology Department
603 E Daniel
Champaign, Illinois 61820

Dr Alan Schoenfeld — 1 copy
Mathematics and Education
The University of Rochester
Rochester, New York 14627

Mr Colin Sheppard — 1 copy
Applied Psychology Unit
Admiralty Marine Technology Est
Teddington, Middlesex
United Kingdom

Dr H Wallace Sinaiko — 1 copy
Program Director
Manpower Research and Advisory Service
Smithsonian Institution
801 North Pitt Street
Alexandria, Virginia 22314

Dr Edward E Smith — 1 copy
Bolt Beranek & Newman
50 Moulton Street
Cambridge, Massachusetts 02138

Dr Richard Snow — 1 copy
School of Education
Stanford University
Stanford, California 94305

Dr Kathryn T Spoehr — 1 copy
Psychology Department
Brown University
Providence, Rhode Island 02912

Dr Gil Ricard — 1 copy
Code N711
NTEC
Orlando, Florida 32813

Dr Worth Scanland — 1 copy
CNET (N-5)
NAS, Pensacola, Florida 32508

Dr Robert G Smith — 1 copy
Office of Chief of Naval Operations
OP-987H
Washington, DC 20350

Dr Alfred F Smode, Director — 1 copy
Training Analysis & Evaluation Group
Department of the Navy
Orlando, Florida 32813

Dr Richard Sorensen — 1 copy
Navy Personnel R&D Center
San Diego, California 92152

Dr Frederick Steinheiser — 1 copy
CNO - OP115
Navy Annex
Arlington, Virginia 20370

Roger Weissinger-Baylon — 1 copy
Department of Administrative Sciences
Naval Postgraduate School
Monterey, California 93940

Mr John H Wolfe — 1 copy
Navy Personnel R&D Center
San Diego, California 92152

Dr Wallace Wulfeck, III — 1 copy
Navy Personnel R&D Center
San Diego, California 92152

Private Sector

Dr John R Anderson — 1 copy
Department of Psychology
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Dr John Annett — 1 copy
Department of Psychology
University of Warwick
Coventry CV4 7AJ
ENGLAND

Dr Michael Atwood — 1 copy
ITT - Programming
1000 Oronoque Lane
Stratford, Connecticut 06497

Dr Alan Baddeley — 1 copy
Medical Research Council
Applied Psychology Unit
15 Chaucer Road
Cambridge CB2 2EF
ENGLAND

Dr Robert Sternberg — 1 copy
Department of Psychology
Yale University
Box 11A, Yale Station
New Haven, Connecticut 06520

Dr Albert Stevens — 1 copy
Bolt Beranek & Newman
10 Moulton Street
Cambridge, Massachusetts 02238

David E Stone, Ph D — 1 copy
Hazeltine Corporation
7680 Old Springhouse Road
McLean, Virginia 22102

Dr Patrick Suppes — 1 copy
Institute for Mathematical Studies in
the Social Sciences
Stanford University
Stanford, California 94305

Dr Kikumi Tatsuoka — 1 copy
Computer Based Education Research Lab
252 Engineering Research Laboratory
Urbana, Illinois 61801

Dr Maurice Tatsuoka — 1 copy
220 Education Bldg
1310 S Sixth Street
Champaign, Illinois 61820

Dr Perry W Thorndyke — 1 copy
Perceptronics, Inc
545 Middlefield Road, Suite 140
Menlo Park, California 94025

Dr Douglas Towne — 1 copy
University of So California
Behavioral Technology Labs
1845 S Elena Avenue
Redondo Beach, California 90277

Dr Kurt Van Lehn — 1 copy
Xerox PARC
3333 Coyote Hill Road
Palo Alto, California 94304

Dr Keith T Wescourt — 1 copy
Perceptronics, Inc
545 Middlefield Road, Suite 140
Menlo Park, California 94025

William B Whitten — 1 copy
Bell Laboratories
2D-610
Holmdel, New Jersey 07733

Dr Mike Williams — 1 copy
Xerox PARC
3333 Coyote Hill Road
Palo Alto, California 94304

Dr Patricia Baggett 1 copy
Department of Psychology
University of Colorado
Boulder, Colorado 80309

Ms Carole A Bagley 1 copy
Minnesota Educational Computing
Consortium
2354 Hidden Valley Lane
Stillwater, Minnesota 55082

Dr Jonathan Baron 1 copy
80 Glenn Avenue
Berwyn, Pennsylvania 19312

Mr Avron Barr 1 copy
Department of Computer Science
Stanford University
Stanford, California 94305

Dr John Black 1 copy
Yale University
Box 11A, Yale Station
New Haven, Connecticut 06520

Dr John S Brown 1 copy
XEROX Palo Alto Research Center
3333 Coyote Road
Palo Alto, California 94304

Dr Bruce Buchanan 1 copy
Department of Computer Science
Stanford University
Stanford, California 94305

Dr Jaime Carbonell 1 copy
Department of Psychology
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Dr Pat Carpenter 1 copy
Department of Psychology
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Dr William Chase 1 copy
Department of Psychology
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Dr Michelene Chi 1 copy
Learning R & D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, Pennsylvania 15213

Civilian Agencies

Dr Patricia A Butler 1 copy
NIE-BRN Bldg, Stop #7
1200 19th Street NW
Washington, DC 20208

Dr Susan Chipman 1 copy
Learning and Development
National Institute of Education
1200 19th Street NW
Washington, DC 20208

Edward Esty 1 copy
Department of Education, OERI
MS 40
1200 19th Street, NW
Washington, DC 20208

Edward J Fuentes 1 copy
Department of Education
1200 19th Street, NW
Washington, DC 20208

TA&E, T&K 1 copy
National Institute of Education
1200 19th Street, NW
Washington, DC 20208

Dr John Mays 1 copy
National Institute of Education
1200 19th Street, NW
Washington, DC 20208

Dr Arthur Melmed 1 copy
724 Brown
U S Dept of Education
Washington, DC 20208

Dr Andrew R Molnar 1 copy
Office of Scientific and Engineering
Personnel and Education
National Science Foundation
Washington, DC 20550

Everett Palmer 1 copy
Research Scientist
Mail Stope 239-3
NASA Ames Research Center
Moffett Field, California 94035

Dr Mary Stoddard 1 copy
C 10, Mail Stop B296
Los Alamos National Laboratories
Los Alamos, New Mexico 87545

Chief, Psychological Research Branch 1 copy
U S Coast Guard (G-P-1/2/TP42)
Washington, DC 20593

Dr William Clancey
Department of Computer Science
Stanford University
Stanford, California 94306
1 copy

Dr Allan M Collins
Bolt Beranek & Newman, Inc
50 Moulton Street
Cambridge, Massachusetts 02138
1 copy

ERIC Facility-Acquisitions
4833 Rugby Avenue
Bethesda, Maryland 20014
1 copy

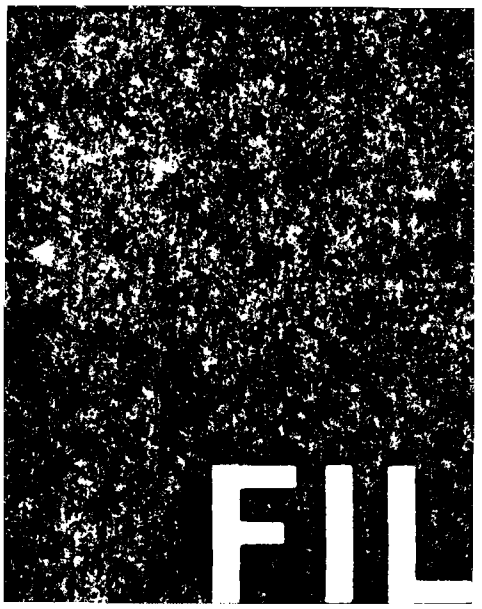
Mr Wallace Feurzeig
Department of Educational Technology
Bolt Beranek and Newman
10 Moulton Street
Cambridge, Massachusetts 02238
1 copy

Dr Dexter Fletcher
WICAT Research Institute
1875 S State Street
Orem, Utah 22333
1 copy

Dr John R Frederiksen
Bolt Beranek & Newman
50 Moulton Street
Cambridge, Massachusetts 02138
1 copy

Dr Frank Withrow
U S Office of Education
400 Maryland Avenue SW
Washington, DC 20202
1 copy

Dr Joseph L Young, Director
Memory & Cognitive Processes
National Science Foundation
Washington, DC 20550
1 copy


FIL